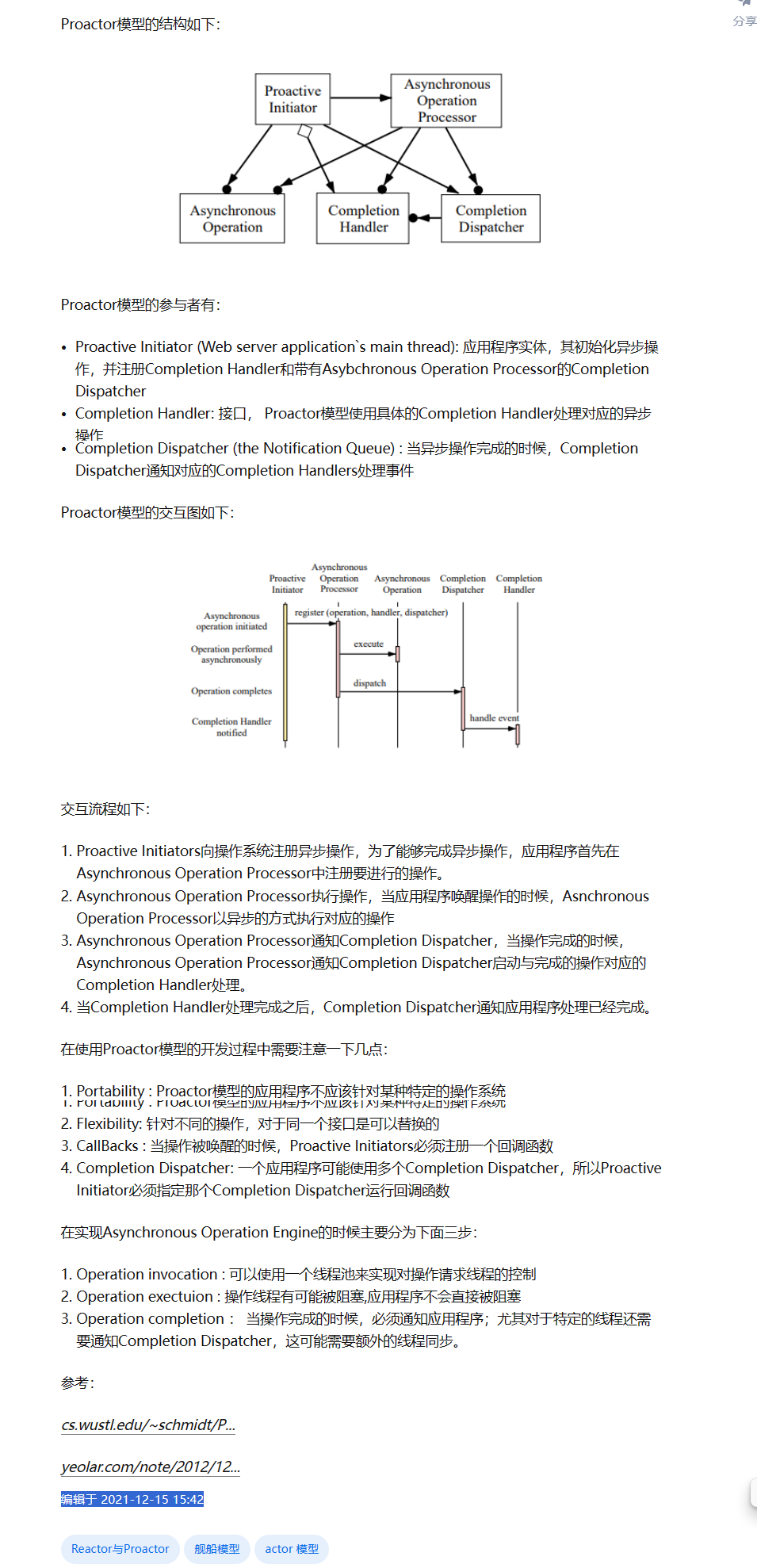
# Proactor模型



# ASIO

https://zhuanlan.zhihu.com/p/583533338

## io\_context

io\_context类为异步I/O对象的用户提供了核心I/O功能，包含：

asio::ip::tcp::socket

asio::ip::tcp::acceptor

asio::ip::udp::socket

asio::deadline\_timer

## 线程安全

对ASIO对象，在一个对象上挂起多个异步操作是安全的；只是指定对该对象的并发调用是不安全的；

## 向io\_context 提交任意任务

向 io\_context 提交函数，请使用 asio::dispatch、asio::post 或 asio::defer 自由函数；

例：

void my\_task()

{

//...

}

int main()

{

asio::io\_context io\_context;

//提交一个函数

asio::post(io\_context, my\_task);

//提交一个lambda 表达式

asio::post(io\_context, [](){

//...

});

//运行 io\_context 直到它用完为止。

io\_context.run();

return 0;

}

## 阻止 io\_context 耗尽工作

## io\_context.run()

当线程调用 io\_context.run() 时，工作和处理程序将从该线程内调用。

asio::io\_context io\_context;

asio::ip::tcp::socket socket(io\_context);

io\_context.post(&print); // 1

socket.connect(endpoint); // 2

socket.async\_receive(buffer, &handle\_async\_receive); // 3

io\_context.post(&print); // 4

io\_context.run(); // 5

std::cout << "Do you reckon this line displays?" << std::endl;

在上面的示例中，io\_context .run()(5) 将阻塞直到：

* 它已从两个print处理程序调用并返回，接收操作成功或失败完成，并且其handle\_async\_receive处理程序已被调用并返回。
* 通过 io\_context 明确停止 io\_context.stop()。
* 从处理程序中抛出异常。

请注意：

1. 当 io\_context 工作用完时，应用程序必须 [r](https://link.zhihu.com/?target=http%3A//www.boost.org/doc/libs/1_56_0/doc/html/boost_asio/reference/io_service/reset.html)eset(), 在 io\_context 再次运行之前。
2. run() 函数会阻塞，直到所有工作完成并且没有更多的处理程序要分派，或者直到 io\_context 停止。

## 工作类对象（asio::io\_context::work）

工作类是一个“在有工作要做时通知 io\_context 的类"。换句话说，只要 io\_context 有一个与之关联的工作对象，它就永远不会无事可做；

#include <iostream>

#include <asio.hpp>

int main()

{

asio::io\_context io\_context;

asio::io\_context::work work(io\_context);

io\_context.run();

std::cout << "Do you reckon this line displays?" << std::endl;

return 0;

}

* 不阻塞工作方式

调用io\_context的poll函数。Poll函数“运行io\_context对象的事件处理循环来执行就绪的处理程序”

#include <iostream>

#include <asio.hpp>

int main()

{

asio::io\_context io\_context;

for(int x = 0; x < 42; ++x)

{

io\_context.poll();

std::cout << "Counter: " << x << std::endl;

}

return 0;

}

* I/O删除一个工作对象

#include <iostream>

#include <asio.hpp>

int main()

{

asio::io\_context io\_context;

//删除此工作对象，必须改用工作对象指针

std::shared\_ptr<asio::io\_context::work> work(new asio::io\_context::work(io\_context));

work.reset();

io\_context.run();

std::cout << "Do you reckon this line displays?" << std::endl;

return 0;

}

# Asio多线程

* 多个线程可以调用 run() 函数来设置一个线程池，io\_context 可以从中执行处理程序。池中等待的所有线程都是等效的，io\_context 可以选择其中的任何一个线程调用处理程序。
* 在多线程的场景下，每个线程都持有一个 io\_context ，并且每个线程都调用各自的 io\_context 的run()方法。
* 全局只分配一个io\_context ，并且让这个 io\_context 在多个线程之间共享，每个线程都调用全局的 io\_context 的run()方法。

## 每个线程一个 I/O Context

应用场景：

1. 在多核的机器上，这种方案可以充分利用多个 CPU 核心。
2. 某个 socket 描述符并不会在多个线程之间共享，所以不需要引入同步机制。
3. 在 event handler 中不能执行阻塞的操作，否则将会阻塞掉 io\_context 所在的线程。

#include <iostream>

#include <asio.hpp>

#include <vector>

class AsioIOContextPool

{

public:

using IOContext = asio::io\_context;

using Work = asio::io\_context::work;

using WorkPtr = std::unique\_ptr<Work>;

//返回当前系统支持的并发线程数

AsioIOContextPool(std::size\_t size = std::thread::hardware\_concurrency()) :

ioContexts\_(size),

works\_(size),

nextIOContext\_(0)

{

for(std::size\_t i = 0; i < size; ++i)

{

works\_[i] = std::unique\_ptr<Work>(new Work(ioContexts\_[i]));

}

for(std::size\_t i = 0; i < ioContexts\_.size(); ++i)

{

threads\_.emplace\_back([this, i](){

ioContexts\_[i].run();

});

}

}

AsioIOContextPool(const AsioIOContextPool&) = delete;

AsioIOContextPool &operator=(const AsioIOContextPool&) = delete;

asio::io\_context& getIOContext()

{

auto &context = ioContexts\_[nextIOContext\_++];

if(nextIOContext\_ == ioContexts\_.size())

{

nextIOContext\_ = 0;

}

return context;

}

void stop()

{

for(auto &work : works\_)

{

work.reset();

}

for(auto &t : threads\_)

{

t.join();

}

}

private:

std::vector<IOContext> ioContexts\_;

std::vector<WorkPtr> works\_;

std::vector<std::thread> threads\_;

std::size\_t nextIOContext\_;

};

int main()

{

std::mutex mtx;

AsioIOContextPool pool;

asio::steady\_timer timer{pool.getIOContext(), std::chrono::seconds{2}};

timer.async\_wait( [&mtx](const asio::error\_code &ec)

{

std::lock\_guard<std::mutex> lock(mtx);

std::cout << "Hello, World!" << std::endl;

}

);

pool.stop();

return 0;

}

## 一个 I/O Service 与多个线程

先分配一个全局 io\_context，然后开启多个线程，每个线程都调用这个io\_context的run()方法。这样，当某个异步事件完成时，io\_context 就会将相应的 event handler 交给任意一个线程去执行。

然而这种方案在实际使用中，需要注意一些问题：

1. 在 event handler 中允许执行阻塞的操作 (例如数据库查询操作)。
2. 线程数可以大于 CPU 核心数，譬如说，如果需要在 event handler 中执行阻塞的操作，为了提高程序的响应速度，这时就需要提高线程的数目。
3. 由于多个线程同时运行事件循环(event loop)，所以会导致一个问题：即一个 socket 描述符可能会在多个线程之间共享，容易出现竞态条件 (race condition)。譬如说，如果某个 socket 的可读事件很快发生了两次，那么就会出现两个线程同时读同一个 socket 的问题 (可以使用strand解决这个问题)。
4. 无锁的同步方式：Asio 提供了 io\_context::strand：如果多个 event handler 通过同一个 strand 对象分发 (dispatch)，那么这些 event handler 就会保证顺序地执行。

#include <iostream>

#include <asio.hpp>

#include <vector>

class AsioThreadPool

{

public:

//返回当前系统支持的并发线程数

AsioThreadPool(std::size\_t size = std::thread::hardware\_concurrency()) :

work\_(new asio::io\_context::work(io\_context\_))

{

for(std::size\_t i = 0; i < size; ++i)

{

threads\_.emplace\_back([this](){

io\_context\_.run();

});

}

}

AsioThreadPool(const AsioThreadPool&) = delete;

AsioThreadPool &operator=(const AsioThreadPool&) = delete;

asio::io\_context& getIOContext()

{

return io\_context\_;

}

void stop()

{

work\_.reset();

for(auto &t : threads\_)

{

t.join();

}

}

private:

asio::io\_context io\_context\_;

std::unique\_ptr<asio::io\_context::work> work\_;

std::vector<std::thread> threads\_;

};

int main()

{

AsioThreadPool pool(4); // 开启 4 个线程

asio::steady\_timer timer1{pool.getIOContext(), std::chrono::seconds{1}};

asio::steady\_timer timer2{pool.getIOContext(), std::chrono::seconds{1}};

int value = 0;

asio::io\_context::strand strand{pool.getIOContext()};

timer1.async\_wait( strand.wrap( [&value] (const asio::error\_code &ec)

{

std::cout << "Hello, World! " << value++ << std::endl;

})

);

timer2.async\_wait(strand.wrap( [&value] (const asio::error\_code &ec)

{

std::cout << "Hello, World! " << value++ << std::endl;

})

);

pool.stop();

return 0;

}